下载APP

21 | 支付前查询订单列表：如何分析优化一个固定的技术组件？

2021-05-07 高楼

高楼的性能工程实战课 进入课程 >

讲述：高楼
时长 18:09 大小 16.64M

今天我们来分析支付前查询订单列表接口。

在这节课中，我将带你来看一下对于一个固定的技术组件，分析优化思路应该是怎样的，也就是说组件不是我们开发的，但是又要分析优化它，我们该怎么办？

此外，我们还会遇到一个问题，就是当数据库的 CPU 并没有全部用完，而是只用了几颗的时候，我们应该如何具体定向？对此，我们将用到查看数据库本身线程栈的方法，这和前面直接看 trx 表有所不同。

下面，我们一起进入今天的内容。

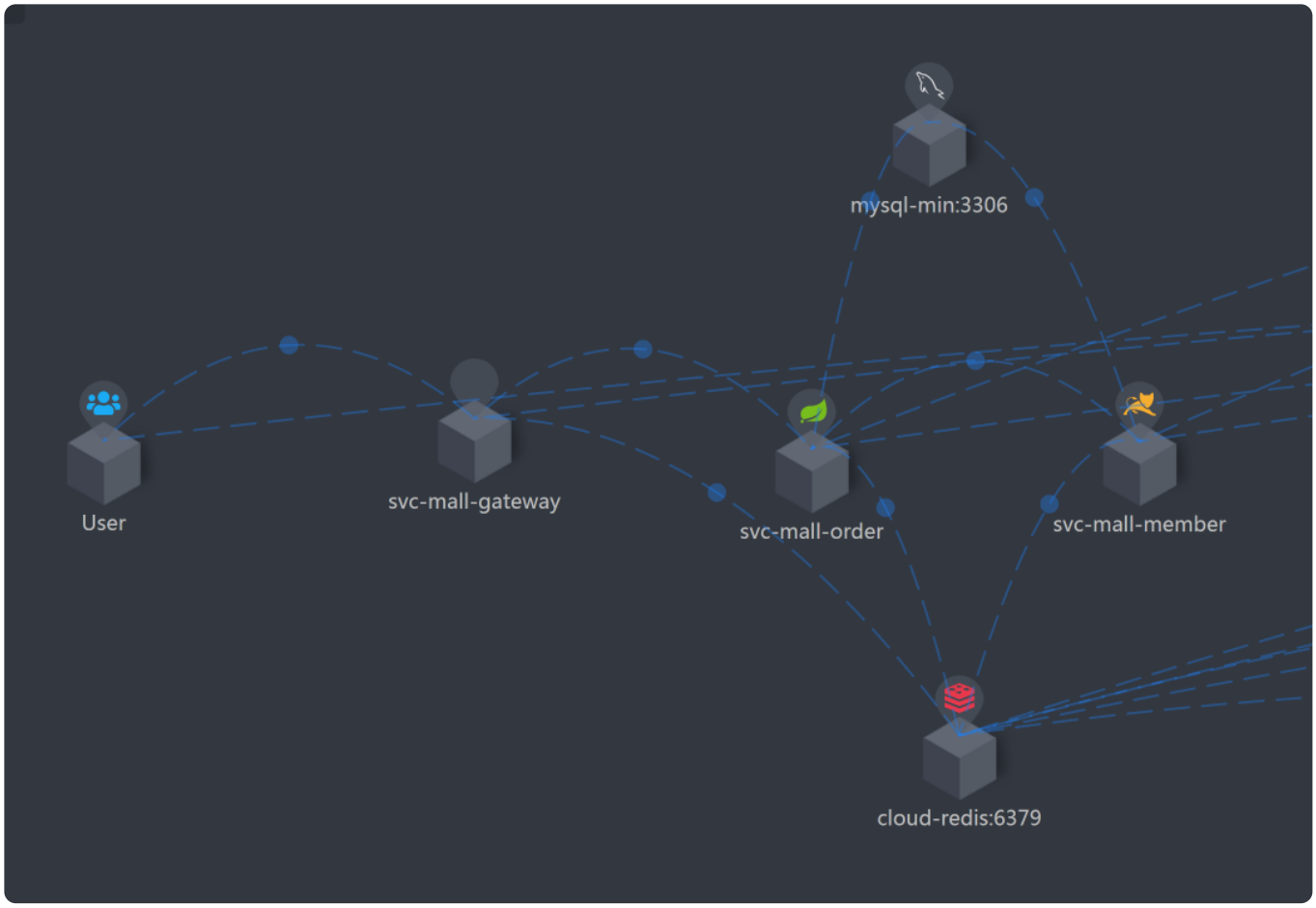
场景运行数据

对于支付前查询订单列表接口，我们先来看第一次运行的性能场景结果：

从运行的场景数据来看，这个接口的 TPS 一开始还是挺高的，达到了 800 多。但是，响应时间也增加了，瓶颈已经出现。我们只要知道瓶颈在哪，就能知道这个接口有没有优化空间。

根据高老师的分析逻辑，在正式分析之前，我们看一下架构图。

架构图

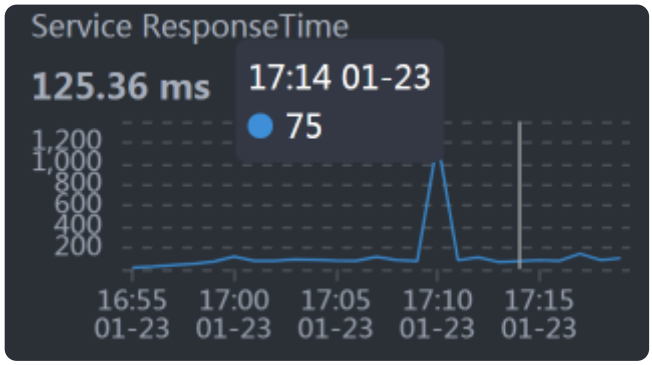


这张架构图是非常清楚的，可以看到，当前接口的逻辑为：Gateway - Order - Member，其中也使用到了 MySQL 和 Redis。

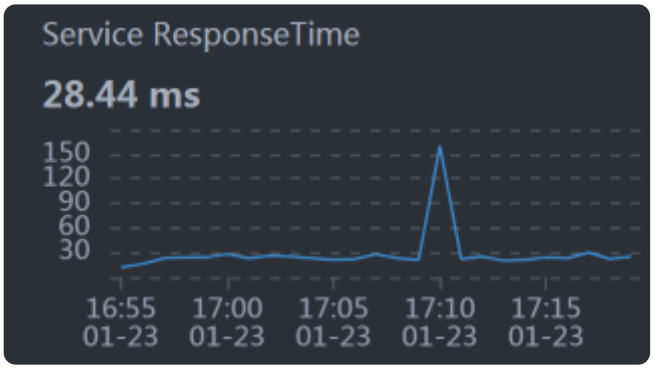
下面我们来看看，响应时间消耗到哪里去了。

拆分响应时间

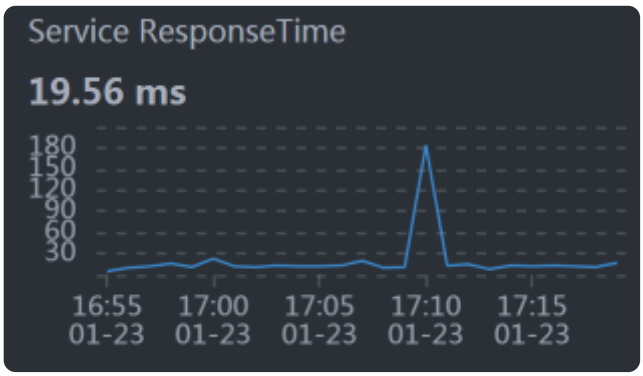
Gateway：



Order：



Member：



从响应时间的分布来看，Gateway（网关）上消耗的时间要长一些。所以，我们接下来得从 Gateway 下手，分析一下到底是哪里消耗了时间。

第一阶段

全局监控分析

按照“**先看全局监控，后看定向监控**”的逻辑，我们先看这个接口的全局监控：

IP（链接到明细）	主机名	运行时间	内存	CPU核	5m负载	CPU使用率	内存使用率	分区使用率*	磁盘读取*	磁盘写入*	连接数	TCP_tw	下载带宽*	上传带宽*
k8s-master-1	k8s-master-1	2.39 week	7.64 GiB	4	0.43	11.20%	24.06%	29.95%	0 B/s	72.78 kB/s	289	25	589.87 kbps	429.27 kbps
k8s-master-2	k8s-master-2	4.98 week	7.64 GiB	4	0.40	16.24%	23.28%	29.95%	0 B/s	71.00 kB/s	288	31	244.39 kbps	328.63 kbps
k8s-master-3	k8s-master-3	4.98 week	7.64 GiB	4	0.34	8.84%	24.92%	29.95%	0 B/s	74.63 kB/s	317	101	849.26 kbps	1.04 Mbps
k8s-worker-1	k8s-worker-1	3.72 day	15.51 GiB	6	1.58	29.60%	44.22%	58.10%	4.60 kB/s	2.35 kB/s	35	74	56.28 Mbps	56.44 Mbps
k8s-worker-2	k8s-worker-2	3.72 day	15.51 GiB	6	0.05	2.67%	14.70%	29.95%	4.60 kB/s	14.96 kB/s	29	78	44.98 kbps	32.76 kbps
k8s-worker-3	k8s-worker-3	1.45 week	15.51 GiB	6	16.80	70.33%	51.69%	48.43%	61.79 kB/s	125.86 kB/s	31	1 K	67.64 Mbps	67.35 Mbps
k8s-worker-4	k8s-worker-4	3.89 week	15.51 GiB	6	12.31	69.54%	36.18%	34.52%	4.60 kB/s	316.58 kB/s	53	24 K	46.56 Mbps	45.98 Mbps
k8s-worker-5	k8s-worker-5	3.89 week	15.51 GiB	6	1.13	15.96%	22.40%	36.40%	9.21 kB/s	3.75 kB/s	28	144	9.02 Mbps	9.33 Mbps
k8s-worker-6	k8s-worker-6	3.89 week	15.51 GiB	6	0.84	13.18%	20.79%	39.27%	13.81 kB/s	3.03 kB/s	29	107	910.98 kbps	870.93 kbps
k8s-worker-7	k8s-worker-7	3.59 week	15.51 GiB	4	0.29	9.32%	28.78%	52.48%	9.21 kB/s	4.20 kB/s	28	129	716.15 kbps	730.72 kbps
k8s-worker-8	k8s-worker-8	2.41 week	15.51 GiB	4	9.72	71.10%	47.91%	51.77%	0 B/s	109.29 kB/s	27	1 K	32.50 Mbps	26.30 Mbps
k8s-worker-9	k8s-worker-9	3.59 week	15.51 GiB	6	2.49	25.16%	25.55%	29.95%	9.21 kB/s	17.13 kB/s	29	73	39.96 Mbps	40.20 Mbps

由于 Gateway 消耗的响应时间长，我们看过全局监控视图之后，要判断出 Gateway 在哪个 worker 上：

复制代码

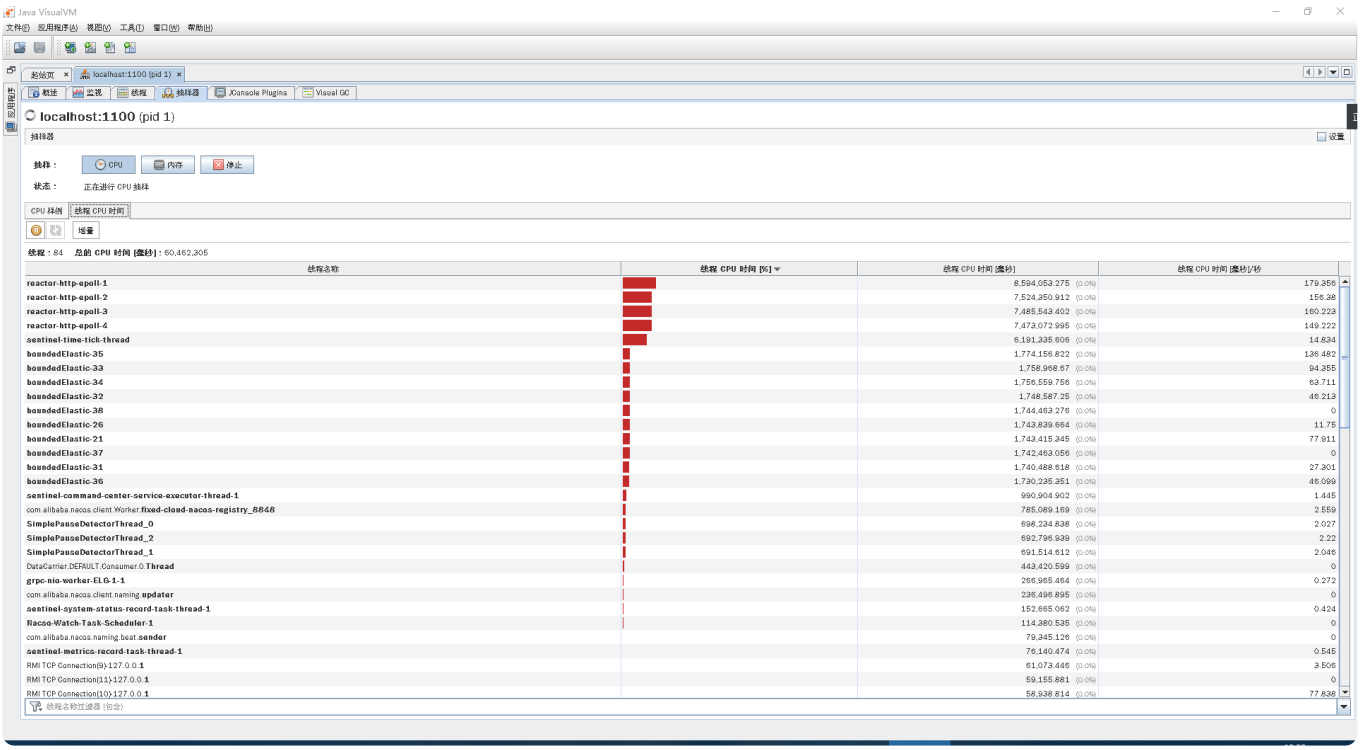
```
1 [root@k8s-master-2 ~]# kubectl get pods -o wide | grep gateway
```

```
gateway-mall-gateway-757659dbc9-tdwnm       1/1     Running     0          3d1
[root@k8s-master-2 ~]#
```

这个 Gateway 服务在 worker-4 上，同时，在全局监控图上可以看到，虽然 Gateway 只消耗了 70% 的 CPU，但它还是消耗了最多的响应时间。既然这样，我们就要关注一下 Gateway 的线程状态，看看它在处理什么。

定向监控分析

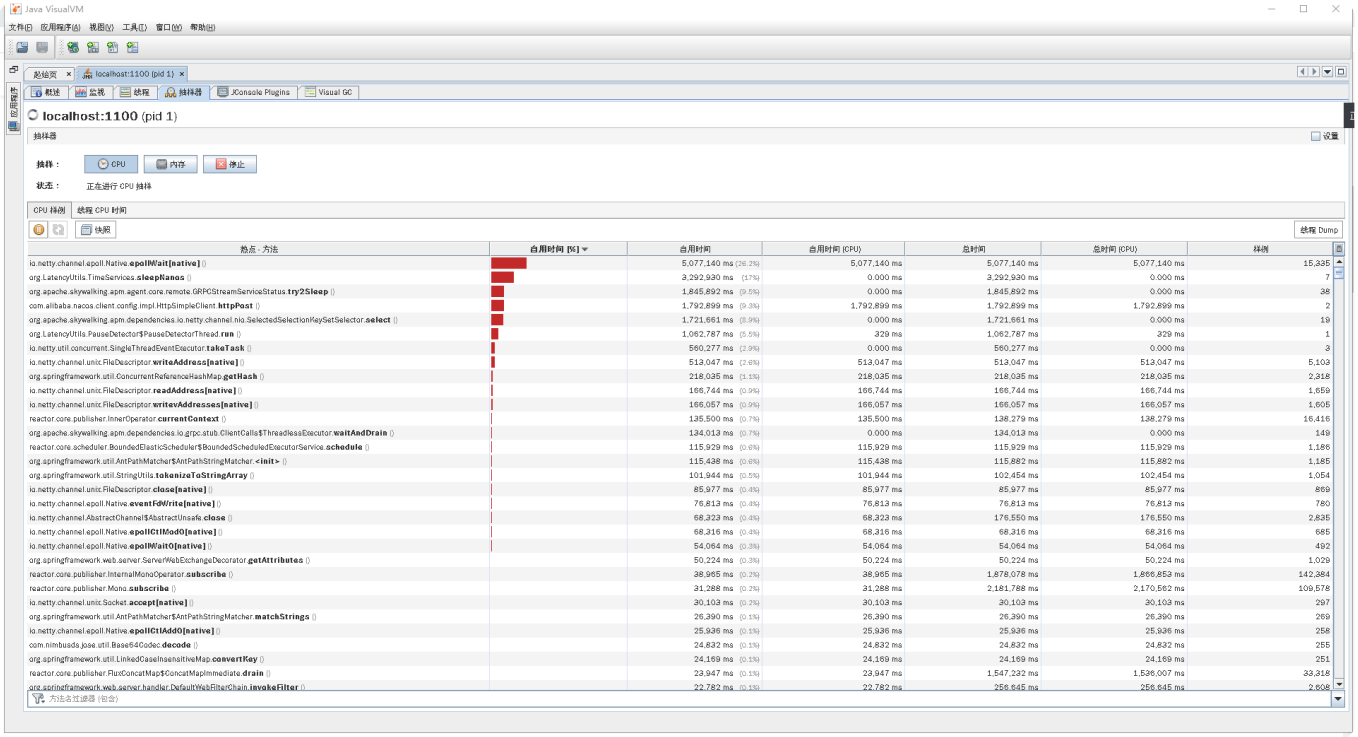
在做定向监控时，我们先来看一下线程的 CPU 消耗：



通过上图可以看到，在 Gateway 中有两类重要的工作线程，分别是 reactor-http-epoll 和 boundedElastic。

在官方的说明中提到，reactor-http-epoll 线程的设置最好与 CPU 个数一致。我们当前的 reactor-http-epoll 线程是 4 个，而这个 worker 有 6C，所以还能增加两个，增加多了意义也不大。至于 boundedElastic，它是有边界的弹性线程池，默认为 CPU 核 x10，也没啥可优化的。

我们再持续看一会儿 Gateway 服务中的线程所消耗的时间比例，看一下方法级的时间消耗有没有异常的情况，也就是比例非常高的，如下图所示：



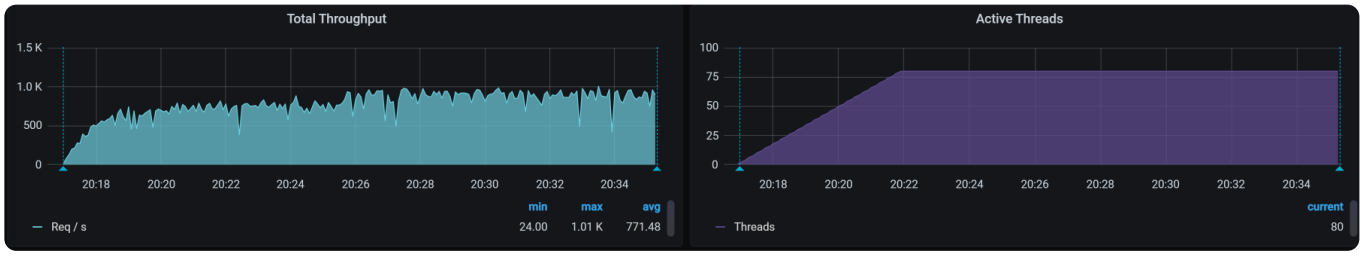
你看，当前的执行方法也都没啥异常的。

现在我们就把线程增加到 6 个，看能不能把 CPU 用高一点。如果 CPU 用多了之后，仍然是 Gateway 消耗的时间长，那我们就只有再继续加 CPU 了。

请你注意，在性能项目中，不要轻易给出加 CPU 这样的建议。一定要在你分析了逻辑之后，确定没有其他优化空间了，再给这样的建议。

优化效果

我们来看一下优化效果：



通过回归测试，我们看到 TPS 有一点增加，只是在图的后半段（由于在测试过程中，Gateway 重启过，前面的 TPS 就当是预热了）增加的并不明显，大概有 50 多 TPS 的样子。不过，也算是有了效果。

我们优化到这里并没有结束，因为在查看各个 Worker 的过程中，我还发现一个奇怪的现象，那就是数据库里有两个 CPU 的使用率非常高。下面我们来扒一扒。

第二阶段

全局监控分析

因为前面优化的效果并不怎么样，所以我们要重新开始分析。让我们从全局监控开始：

IP（链接到明细）	主机名	运行时间	内存	CPU核	5m负载	CPU使用率	内存使用率	分区使用率*	磁盘读取*	磁盘写入*	连接数	TCP_tw	下载带宽*	上传带宽*
k8s-master-1	k8s-master-1	2.39 week	7.64 GiB	4	0.42	11.12%	24.05%	29.95%	0 B/s	72.44 kB/s	289	27	589.20 kbps	426.69 kbps
k8s-master-2	k8s-master-2	4.98 week	7.64 GiB	4	0.47	16.09%	23.27%	29.95%	0 B/s	69.38 kB/s	289	28	242.92 kbps	323.29 kbps
k8s-master-3	k8s-master-3	4.98 week	7.64 GiB	4	0.31	8.77%	24.95%	29.95%	0 B/s	73.38 kB/s	317	104	834.88 kbps	1.03 Mbps
k8s-worker-1	k8s-worker-1	3.72 day	15.51 GiB	6	1.57	29.28%	44.23%	58.10%	4.60 kB/s	10.49 kB/s	36	74	54.97 Mbps	55.13 Mbps
k8s-worker-2	k8s-worker-2	3.72 day	15.51 GiB	6	0.04	2.64%	14.70%	29.95%	0 B/s	629.03 B/s	29	77	44.73 kbps	32.79 kbps
k8s-worker-3	k8s-worker-3	1.45 week	15.51 GiB	6	16.87	69.00%	51.63%	48.43%	22.43 kB/s	160.40 kB/s	31	1 K	66.19 Mbps	65.91 Mbps
k8s-worker-4	k8s-worker-4	3.89 week	15.51 GiB	6	12.36	68.42%	36.19%	34.52%	9.21 kB/s	267.20 kB/s	53	24 K	45.79 Mbps	45.24 Mbps
k8s-worker-5	k8s-worker-5	3.89 week	15.51 GiB	6	1.14	16.53%	22.40%	36.40%	13.81 kB/s	3.54 kB/s	28	141	8.81 Mbps	9.12 Mbps
k8s-worker-6	k8s-worker-6	3.89 week	15.51 GiB	6	0.79	14.06%	20.66%	39.27%	13.81 kB/s	2.40 kB/s	29	122	908.29 kbps	870.25 kbps
k8s-worker-7	k8s-worker-7	3.59 week	15.51 GiB	4	0.28	9.08%	28.81%	51.42%	9.21 kB/s	4.39 kB/s	28	133	784.05 kbps	796.68 kbps
k8s-worker-8	k8s-worker-8	2.41 week	15.51 GiB	4	9.47	72.40%	47.90%	51.77%	0 B/s	98.29 kB/s	27	1 K	33.36 Mbps	25.71 Mbps
k8s-worker-9	k8s-worker-9	3.59 week	15.51 GiB	6	2.37	25.00%	25.56%	29.95%	4.60 kB/s	14.47 kB/s	29	71	36.73 Mbps	41.13 Mbps

看起来倒是没啥，数据库所在的 worker-1 也不见有什么大的资源消耗。

请你注意，我在文章中经常用这个界面来看全局监控的数据。但这并不是说，我只看这个界面。当我在这个界面中看不到明显的问题点时，我也会去看一些命令，像 top/vmstat 等，这和我一直说的全局监控的完整计数器有关。因此，你的脑袋里要有全局监控计数器的视图，然后才能真正看全第一层的计数器。

我们再来看数据库所在的 worker 上的 top 数据，发现了这样的现象：

复制代码

```
bash-4.2$ top
top - 09:57:43 up 3 days, 17:54,  0 users,  load average: 4.40, 3.57, 3.11
Tasks:  11 total,   1 running,   9 sleeping,   1 stopped,   0 zombie
%Cpu0  :  8.0 us,  4.7 sy,  0.0 ni, 84.3 id,  0.0 wa,  0.0 hi,  2.2 si,  0.7 s
%Cpu1  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 s
%Cpu2  :  6.5 us,  4.4 sy,  0.0 ni, 85.5 id,  0.0 wa,  0.0 hi,  2.2 si,  1.5 s
%Cpu3  :  7.8 us,  5.7 sy,  0.0 ni, 83.7 id,  0.0 wa,  0.0 hi,  2.1 si,  0.7 s
%Cpu4  : 96.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  4.0 si,  0.0 s
%Cpu5  :  7.0 us,  4.0 sy,  0.0 ni, 84.9 id,  0.0 wa,  0.0 hi,  2.6 si,  1.5 s
KiB Mem : 16265992 total,  1203032 free,  6695156 used,  8367804 buff/cache
KiB Swap:        0 total,        0 free,        0 used.  9050344 avail Mem


  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
```

```
15      1 mysql     20   0 8272536   4.7g  13196 S 248.8 30.5   6184:36 mysqld
```

可以非常明显地看到，有两个 CPU 的使用率高，那我们就来定向分析下数据库。

在此之前，我们不妨心中默念 10 遍“只要思路不乱，任何问题都是一盘菜”，因为保持思路清晰非常重要。

定向监控分析

我们要定向分析数据库，可是在数据库上又不是所有的 CPU 使用率都高，所以，我们要来看一下数据库线程到底在做什么动作。有了上面的进程信息之后，我们再深入到线程级：

复制代码

```
1  bash-4.2$ top -Hp 1
2  top - 09:56:40 up 3 days, 17:53,  0 users,  load average: 3.05, 3.30, 3.01
3  Threads:  92 total,   2 running,  90 sleeping,   0 stopped,   0 zombie
4  %Cpu0  :  5.4 us,  2.9 sy,  0.0 ni, 89.2 id,  0.0 wa,  0.0 hi,  2.2 si,  0.4 s
5  %Cpu1  : 99.7 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.3 s
6  %Cpu2  :  5.4 us,  3.2 sy,  0.0 ni, 88.2 id,  0.0 wa,  0.0 hi,  2.5 si,  0.7 s
7  %Cpu3  :  6.3 us,  4.2 sy,  0.0 ni, 87.0 id,  0.0 wa,  0.0 hi,  2.1 si,  0.4 s
8  %Cpu4  : 96.3 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  3.7 si,  0.0 s
9  %Cpu5  :  4.0 us,  2.5 sy,  0.0 ni, 91.0 id,  0.0 wa,  0.0 hi,  1.8 si,  0.7 s
10 KiB Mem : 16265992 total,  1205356 free,  6692736 used,  8367900 buff/cache
11 KiB Swap:        0 total,        0 free,        0 used.  9052664 avail Mem
12
13
14   PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
15   311 mysql     20   0 8272536   4.7g  13196 R 99.9 30.5  18:20.34 mysqld
16   241 mysql     20   0 8272536   4.7g  13196 R 99.7 30.5   1906:40 mysqld
17   291 mysql     20   0 8272536   4.7g  13196 S  3.3 30.5  15:49.21 mysqld
18   319 mysql     20   0 8272536   4.7g  13196 S  3.0 30.5  11:50.34 mysqld
19   355 mysql     20   0 8272536   4.7g  13196 S  3.0 30.5  13:01.53 mysqld
20   265 mysql     20   0 8272536   4.7g  13196 S  2.7 30.5  18:17.48 mysqld
21   307 mysql     20   0 8272536   4.7g  13196 S  2.7 30.5  16:47.77 mysqld
22   328 mysql     20   0 8272536   4.7g  13196 S  2.7 30.5  15:34.92 mysqld
23   335 mysql     20   0 8272536   4.7g  13196 S  2.7 30.5   8:55.38 mysqld
24   316 mysql     20   0 8272536   4.7g  13196 S  2.3 30.5  14:38.68 mysqld
25   350 mysql     20   0 8272536   4.7g  13196 S  2.3 30.5  10:37.94 mysqld
26   233 mysql     20   0 8272536   4.7g  13196 S  2.0 30.5  14:19.32 mysqld
27   279 mysql     20   0 8272536   4.7g  13196 S  2.0 30.5  19:51.80 mysqld
28   318 mysql     20   0 8272536   4.7g  13196 S  2.0 30.5  11:34.62 mysqld
29   331 mysql     20   0 8272536   4.7g  13196 S  2.0 30.5  11:46.94 mysqld
30   375 mysql     20   0 8272536   4.7g  13196 S  2.0 30.5   1:29.22 mysqld
31   300 mysql     20   0 8272536   4.7g  13196 S  1.7 30.5  17:45.26 mysqld
32   380 mysql     20   0 8272536   4.7g  13196 S  1.7 30.5   1:24.32 mysqld
```

你看，只有两个 MySQL 的线程在使用 CPU。到了这一步，你可能会想，接下来去查 SQL！虽然可能就是 SQL 的问题，但我还是建议你**找到相应的证据。**

由于 MySQL 是用 C 语言写的，那我们就用 gstack（这是一个装了 GDB 之后就会有的命令）打印一下这两个 MySQL 的栈看看具体的函数。我们把那两个 PID（311、241）的栈拿出来之后，看到如下信息：

复制代码

```
Thread 59 (Thread 0x7f1d60174700 (LWP 241)):
#0  0x000055a431fefea9 in JOIN_CACHE::read_record_field(st_cache_field*, bool)
#1  0x000055a431ff01ca in JOIN_CACHE::read_some_record_fields() ()
#2  0x000055a431ff070f in JOIN_CACHE::get_record() ()
#3  0x000055a431ff2a92 in JOIN_CACHE_BNL::join_matching_records(bool) ()
#4  0x000055a431ff18f0 in JOIN_CACHE::join_records(bool) ()
#5  0x000055a431e397c0 in evaluate_join_record(JOIN*, QEP_TAB*) ()
#6  0x000055a431e3f1a5 in sub_select(JOIN*, QEP_TAB*, bool) ()
#7  0x000055a431e37a90 in JOIN::exec() ()
#8  0x000055a431eaa0ba in handle_query(THD*, LEX*, Query_result*, unsigned lon
#9  0x000055a43194760d in execute_sqlcom_select(THD*, TABLE_LIST*) ()
#10 0x000055a431e6accf in mysql_execute_command(THD*, bool) ()
#11 0x000055a431e6d455 in mysql_parse(THD*, Parser_state*) ()
#12 0x000055a431e6e3b6 in dispatch_command(THD*, COM_DATA const*, enum_server_
#13 0x000055a431e6fc00 in do_command(THD*) ()
#14 0x000055a431f33938 in handle_connection ()
#15 0x000055a4320e66d4 in pfs_spawn_thread ()
#16 0x00007f1e8f1fcdd5 in start_thread () from /lib64/libpthread.so.0
#17 0x00007f1e8d3cc02d in clone () from /lib64/libc.so.6
Thread 41 (Thread 0x7f1d585e0700 (LWP 311)):
#0  0x000055a4319dbe44 in Item_field::val_int() ()
#1  0x000055a4319fb839 in Arg_comparator::compare_int_signed() ()
#2  0x000055a4319fbd9b in Item_func_eq::val_int() ()
#3  0x000055a431ff24ab in JOIN_CACHE::check_match(unsigned char*) ()
#4  0x000055a431ff26ec in JOIN_CACHE::generate_full_extensions(unsigned char*)
#5  0x000055a431ff2ab4 in JOIN_CACHE_BNL::join_matching_records(bool) ()
#6  0x000055a431ff18f0 in JOIN_CACHE::join_records(bool) ()
#7  0x000055a431e397c0 in evaluate_join_record(JOIN*, QEP_TAB*) ()
#8  0x000055a431e3f1a5 in sub_select(JOIN*, QEP_TAB*, bool) ()
#9  0x000055a431e37a90 in JOIN::exec() ()
#10 0x000055a431eaa0ba in handle_query(THD*, LEX*, Query_result*, unsigned lon
#11 0x000055a43194760d in execute_sqlcom_select(THD*, TABLE_LIST*) ()
#12 0x000055a431e6accf in mysql_execute_command(THD*, bool) ()
#13 0x000055a431e6d455 in mysql_parse(THD*, Parser_state*) ()
#14 0x000055a431e6e3b6 in dispatch_command(THD*, COM_DATA const*, enum_server_
#15 0x000055a431e6fc00 in do_command(THD*) ()
#16 0x000055a431f33938 in handle_connection ()
#17 0x000055a4320e66d4 in pfs_spawn_thread ()
#18 0x00007f1e8f1fcdd5 in start_thread () from /lib64/libpthread.so.0
#19 0x00007f1e8d3cc02d in clone () from /lib64/libc.so.6
```

很明显，是两个 execute_sqlcom_select 函数，也就是两个 select 语句。我们接着往上看栈，还可以看到是 JOIN 函数。既然是 select 语句中的 JOIN，那我们直接去找 SQL 语句就好了。

因此，我们直接去查 innodb_trx 表，看看正在执行 SQL 有没有消耗时间长的。你也许会执行 show processlist 之类的命令，但是为了看全 SQL，我还是建议你直接查 trx 表。由于我们使用的 thread_handling 是默认的 one-thread-per-connection，操作系统的线程和 mysql 里的线程都是一一对应的。所以，我们在这里直接查 trx 表不会有什么误判。

通过查找 innodb_trx 表，我们看到了这样两个 SQL 消耗时间较长，列在这里：

复制代码

```
-- sql1
SELECT
  count(*)
FROM
  oms_order o
LEFT JOIN oms_order_item ot ON o.id = ot.order_id
WHERE
  o. STATUS = 0
AND o.create_time < date_add(NOW(), INTERVAL - 120 MINUTE)
LIMIT 0,
 1000


-- sql2:
SELECT
  o.id,
  o.order_sn,
  o.coupon_id,
  o.integration,
  o.member_id,
  o.use_integration,
  ot.id ot_id,
  ot.product_name ot_product_name,
  ot.product_sku_id ot_product_sku_id,
  ot.product_sku_code ot_product_sku_code,
  ot.product_quantity ot_product_quantity
FROM
  oms_order o
LEFT JOIN oms_order_item ot ON o.id = ot.order_id
WHERE
  o. STATUS = 0
```

```
AND o.create_time < date_add(NOW(), INTERVAL - 120 MINUTE)
```

我们提到多次，要想看 SQL 慢，就得看 SQL 对应的执行计划（在 MySQL 中，如果执行计划看得不清楚，还可以看 Profile 信息）。这两个 SQL 对应的执行计划如下：

```
EXPLAIN SELECT
  o.id,
  o.order_sn,
  o.coupon_id,
  o.integration,
  o.member_id,
  o.use_integration,
  ot.id ot_id,
  ot.product_name ot_product_name,
  ot.product_sku_id ot_product_sku_id,
  ot.product_sku_code ot_product_sku_code,
  ot.product_quantity ot_product_quantity
FROM
  oms_order o
LEFT JOIN oms_order_item ot ON o.id = ot.order_id
WHERE
  o. STATUS = 0
AND o.create_time < date_add(NOW(), INTERVAL - 120 MINUTE)
```

信息 | 结果1 | 概况 | 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	o	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	12476208	3.33	Using wh
1	SIMPLE	ot	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	170561	100	Using wh

```
EXPLAIN SELECT
  count(*)
FROM
  oms_order o
LEFT JOIN oms_order_item ot ON o.id = ot.order_id
WHERE
  o. STATUS = 0
AND o.create_time < date_add(NOW(), INTERVAL - 120 MINUTE)
LIMIT 0,
 1000
```

息 | 结果1 | 概况 | 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	o	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	12476208	3.33	Using where
1	SIMPLE	ot	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	170561	100	Using where; Using join buffer (Block Nestec

依然是常见的全表扫描。看到这里，你是不是有一种索然无味的感觉？但是，我们还是需要知道这两个语句为什么会产生。

其实，支付前查询订单列表这个接口并没有用到这两个 SQL。于是，我到代码中看了一下这两个 SQL 的生成过程，反向查找到如下代码：

复制代码

```
@Scheduled(cron = "0 0/20 * ? * ?")
    private void cancelTimeOutOrder(){
        Integer count = portalOrderService.cancelTimeOutOrder();
        LOGGER.info("取消订单释放锁定库存：{}",count);
    }
```

很显然，这是一个定时计划，每 20 分钟执行一次。到这里，问题就很清楚了，原来是定时任务调用了这两个批量的查询语句，导致了两个 CPU 使用率达到 100%，并且也持续了一段时间。

像这样的定时任务，我们要格外关注一下，注意把它和实时业务分开部署和处理，减少批量业务对实时业务的资源争用。如果放在一起处理，那就要控制好要批量查询的数据量级，让 SQL 的查询变得合理。

由于数据库可用的 CPU 比较多，这个定时任务对我们的 TPS 并没有产生什么明显的影响，在这里我们不用做什么处理，以后注意分开就好了。

总结

在这节课中，虽然我们的优化并没有让 TPS 明显增加，但是因为分析的技术细节不一样，我也非常完整地记录了整个分析过程。

在第一阶段的分析中，我们运用的还是之前提到的分析思路。不同点在于，对于一个非常成熟的固定组件，我们要想优化它，就要去了解它的架构，找到它的相关性能参数。因为在实际的性能项目中，面对这样的组件，我们往往没有时间去纠结内部的实现，需要非常快速地作出判断。如果时间允许，你倒是可以慢慢折腾。

其实理解一个技术组件的原理，并没有想像中的那么高不可攀、深不可测，只要耐心看下去，你总会成长。

在第二阶段的分析中，我们由某几个 CPU 高的现象分析到了具体的 SQL 问题。这个过程虽然简单，但是从这个问题上，我们可以看出这个系统还有很多的优化空间，比如说主从分离、定时任务拆为单独的服务等等。不过，在我们的性能分析中，重点仍然是我跟你一直灌输的分析思路，希望你记在心里了。

课后作业

最后，我给你留两道题，请你思考一下：

1. 为什么要看全部的全局监控计数器？

2. 单 CPU 高时，如何定位具体的问题点？你有什么思路？

记得在留言区和我讨论、交流你的想法，每一次思考都会让你更进一步。

如果你读完这篇文章有所收获，也欢迎你分享给你的朋友，共同学习进步。我们下一讲再见！

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 20 | 生成订单信息之二：业务逻辑复杂，怎么做性能优化?

下一篇 22 | 支付订单信息：如何高效解决for循环产生的内存溢出?

精选留言 (2)

写留言

道长
2021-05-12

1、也可能存在关联系统影响导致性能差；
2、CPU使用率高，也可以使用perf抓去调用函数

展开

作者回复: 1，在性能分析中没有“可能”。其实这个问题要问的是一个思路，而不是原因。
2. 也同样是问思路。不过cpu使用率，应该先看是哪个计数器，再确定后续的思路，而不是直接用perf去看系统调用。

alley
2021-05-07

老师，我看到一句话，说一切性能问题，都能通过jstack pid找到线索

展开

作者回复: 那要看是不是和代码有关的问题喽。
jstack是查看栈信息的，也就是代码的执行逻辑。